Lecture 7: Sequence types and control flow

In our previous lecture, we introduced a few basic commands and concepts in Python 🐍 , i.e. printing and storing values in variables, and two important data types: **numbers** (integers, floats) and **strings**.

In this lecture, we introduce additional data types and another important concept: **Control Flow**

> In programming, **Control Flow** is the order in which individual statements, commands, or function calls are evaluated.

Consider the example of computing the cost of a basket that we have seen last week:

```python
# Compute cost of individual goods and store them in a variables
cost_cola      = 1.25 * 5
cost_water     = 0.75 * 7
cost_chocolate = 3 * 4

# Use above variables to store the cost of the basket
cost_basket = cost_cola + cost_water + cost_chocolate

# Display the result to screen
print(cost_basket)
```

When the above code is run, Python evaluates the above operations, variable assignments, and function calls ***sequentially***.

However, there are cases in which the **number** and **correct order** of instructions is not as simple as the one above. For instance, you might have that

- Your shopping list is quite long, say 100+ products, which requires you to type all computations by hand
- The quantity to buy might depend on the price of the good

In such cases, Python have built-in statements that allow us to regulate a program's control flow:

- **Loops**
- **Conditional statements**

We will start working with these during this week's lecture.

Sequence types: lists, tuples, and dictionaries

Before we discuss loops and conditional statements, we need introduce additional data types. As we will see, these data types are essential when working with control flow statements.

We know how to store information in single variables. Numbers and text can be assigned to a variable using the two data types we introduced above: **Numbers** and **Strings**.

That's great but what if we have lots of them?

Sometimes it is useful to store information in a single place. This can result in a tidier code and less usage of many redundant variables.

Python can store multiples numbers and strings in so-called **sequence types**

- **Lists**
- **Tuples**
- **Dictionaries**

Let's introduce them!

Lists

In Python, a **list** is a collection of objects enclosed by square brackets `[]` and separated by commas `,`.

Lists can be created and stored in a variable `list_name` using the following syntax:

```
list_name = [first_object, second_objects, ... , last_object]
```

As mentioned above, lists are useful when we have many values that we want to store in a single place.

Creating lists

Let's revisit the example of computing the cost of a basket once again. All relevant information is contained in the table below:

Good	Price	Quantity
Cola	1.25	5
Water	0.75	7
Chocolate	3	4

The aim is to store all relevant information in fewer variables than we can access later on. Here's two approaches to do so:

- **Approach #1**: Group prices and quantities according to the goods purchased
- **Approach #2**: Group all prices and all quantities separately

Let's see these approaches in action.

Approach #1

In [1]:

```
# For each good, create a list containing the price and quantity purchased
cola      = [1.25, 5]
water     = [0.75, 7]
chocolate = [3, 4]

# Display prices and quantities for each good
print(cola)
print(water)
print(chocolate)
```

```
[1.25, 5]
[0.75, 7]
[3, 4]
```

Approach #2

In [2]:

```
# Create a list with all prices
prices = [1.25, 0.75, 3]

# Create a list with all quantities
quantities = [5, 7, 4]

# Display prices and quantities
print(prices)
print(quantities)
```

```
[1.25, 0.75, 3]
[5, 7, 4]
```

Accessing lists elements: indexing and slicing

Now that we have stored all information in lists, we want to be able to access them.

Similarly to strings, we can do that with **indexing** and **slicing.** The syntax for accessing individual or subsets of elements is identical to the one that you would use with strings.

Indexing

Given a list `list_name` , we can access the element in position `index` with

```
list_name[index]
```

Slicing

Given a list `list_name` , we can access the elements from `start` (included) to `end` (**not** included) with

```
list_name[start:end]
```

This will allow us to access the individual elements, i.e. prices and quantities, and compute the cost of the basket.

In [3]:

```
# Compute the cost of the basket
# Approach #1
cost_basket1 = cola[0]*cola[1] + water[0]*water[1] + chocolate[0]*chocolate[1]
print(cost_basket1)

# Approach #2
cost_basket2 = prices[0]*quantities[0] + prices[1]*quantities[1] + prices[2]*qua
ntities[2]
print(cost_basket2)
```

```
23.5
23.5
```

What can be stored in a list? Well, everything!

Lists are a super-flexible objects. Indeed, lists can contain all data types in Python including numbers, strings, and much more.

Here's an example:

In [4]:

```
# Define a list containing numbers and strings
many_types_list = [12, "Ciao", 3.14, "Pizza".upper()]
print(many_types_list)
```

```
[12, 'Ciao', 3.14, 'PIZZA']
```

Lists can even contain lists themselves! 🤯

> Lists that contain other lists are called **nested lists.**

They allows us to store all information related to our basket in one variable:

In [5]:

```
# Store all information in one variable
basket = [cola, water, chocolate]
print(basket)
```

```
[[1.25, 5], [0.75, 7], [3, 4]]
```

We can then access the values we are interested in by indexing the list **sequentially**:

```
list_name[outer_list_index][inner_list_index]
```

In [6]:

```
# Get the price of chocolate
print(f"The price of chocolate is £ {basket[-1][0]}")
```

```
The price of chocolate is £ 3
```

Working with lists #1: updating lists

Lists are **mutable** objects. This means that the elements of a list can be updated with new information if necessary.

This allow us to reflect changes in the **existing** list without having to create a new list.

The syntax to update elements in a list is

```
list_name[old_value_index] = new_value
```

For instance, let's say that the price of cola has changed from £1.25 to £1.50. We can update it as follows:

In [7]:

```
# Display old value
print(f"The price of cola was £{cola[0]}")

# Update value
cola[0] = 1.5

# Display new value
print(f"The price of cola is now £{cola[0]}")
```

```
The price of cola was £1.25
The price of cola is now £1.5
```

Working with lists #2: list methods

When working with lists we are not limited to updating single elements. We can add new elements, delete existing elements, combine two lists into one, and more. This is done using **list methods**.

Similarly to strings methods, a list method is a command that allow us to transform the existing list.

The general syntax for calling a list method is:

```
list_name.method_name(<arguments>)
```

A complete list of list methods can be found in the official Python documentation here (https://docs.python.org/3/tutorial/datastructures.html).

Updating goods in the basket

You arrived at the self checkout and suddenly you changed your mind on the chocolate. You fancy crisps instead!

Therefore, you want to update your basket by **adding** crisps and **dropping** chocolate.

Imagine that you have a physical basket with you. You would do two things:

1. First you will add the bags of crisps in the basket
    - This is done with the `list.append()` method
2. Then you will remove the chocolate from it
    - This is done with the `del` statement

In [8]:

```
# Store the price and quantity of crisps in a list
crisps = [1.95, 3]

# Add the crisps to your basket list
basket.append(crisps)

# Check that the crisps are there
print(basket)

# Take chocolate out of the basket (you need to know where it is)
del basket[-2]

# Check that the basket contains what you want
print(basket)
```

```
[[1.5, 5], [0.75, 7], [3, 4], [1.95, 3]]
[[1.5, 5], [0.75, 7], [1.95, 3]]
```

Tuples

A **tuple** is another data type that can represent a sequence of elements.

Tuples are very similar to lists, except they are **immutable**. Once they are created, they **cannot** be modified.

Tuples are defined by comma-separated values enclosed in round brackets `()` according to the following syntax

```
tuple_name = (first_object, second_objects, ... , last_object)
```

where round brackets `()` may be omitted in some cases.

As with lists, tuples can contain different data types. For instance:

In [9]:

```python
# Define a tuple with different data types
my_tuple = (1, "Ciao", [20, "Nov", 2020])
print(my_tuple)
```

```
(1, 'Ciao', [20, 'Nov', 2020])
```

Unpacking

Tuples can be very handy when we want to assign different variables to individual elements of a sequence. This operation is called **unpacking**:

In [10]:

```python
# Assign the elements of my_tuple to individual variables
a, b, c = my_tuple
print(f"The value of a is {a}")
print(f"The value of b is {b}")
print(f"The value of c is {c}")
```

```
The value of a is 1
The value of b is Ciao
The value of c is [20, 'Nov', 2020]
```

Dictionaries

Another data type that can represent a collection of items is a **dictionary**.

> A **dictionary** is a comma-separated list of `key: value` pairs that allow us to store a collection of **values** and associated **keys**.

More precisely, a dictionary is a **mapping type** since it associates (maps) keys into values.

Dictionaries are defined by a comma-separated list of keys and values enclosed in curly brackets `{}`. That is

```
dict_name = {key1: value1, key2: value2, ... , keyN: valueN}
```

where

- The `key` must be an **immutable** object
- The `value` can be any data type

Dictionaries can be defined on a multiple lines as well. See an example below:

```
In [11]:
```

```python
# Define a dictionary containing first name, surname, and age of your lecturer
Luca = {
        "firstname": "Luca",
        "surname": "Rondina",
        "age": 30
       }

print(Luca)
```

```
{'firstname': 'Luca', 'surname': 'Rondina', 'age': 30}
```

Dictionaries are well suited to those cases where we want to store and access information with different interpretation but related to the same case. In the example above, each value has a different interpretation but all values refer to one person.

Working with dictionaries #1: accessing values through keys

A dictionary `value` can be accessed by using the correspondent `key` with the following syntax:

```
dict_name[key]
```

Notice the difference between lists and dictionaries:

- Lists elements are **ordered** and are associated to an **positional index**
- Dictionary values are **unordered** and are associated to a **key**

```
In [12]:
```

```python
# Display the age of your lecturer
print(Luca['age'])
```

```
30
```

As with lists, dictionary values can be updated.

```
In [13]:
```

```python
# Suppose we are in 2021: update the age value
Luca['age'] = 31
print(Luca)
```

```
{'firstname': 'Luca', 'surname': 'Rondina', 'age': 31}
```

Nested dictionaries

Similarly to lists, we can have a **nested dictionary**, that is a dictionary whose values are dictionary themselves.

Our shopping basket is a perfect example of a good use of a nested dictionary:

In [14]:

```
# Redefine the basket variable as a nested dictionary
basket = {
          "cola"     :  {"price": 1.25, "quantity": 5},
          "water"    :  {"price": 0.75, "quantity": 7},
          "chocolate":  {"price": 3   , "quantity": 4},
}

# Get and display the price of water
price_water = basket['water']['price']
print(price_water)
```

```
0.75
```

Working with dictionaries #2: dictionary methods

Similarly to lists, dictionaries come with built-in methods that allows us to update, add, and delete values in a dictionary as well as perform other operations.

Some of the most used methods are:

Method name	Method description	Usage
clear()	Deletes everything from a dictionary and leaves an empty dictionary	`dict_name.clear()`
get()	Returns the value of key. If key is not found it returns `None` , instead of throwing `KeyError` exception	`dict_name.get(<key>)`
pop()	Removes the item from the dictionary and returns its value as output. If the key is not found, `KeyError` will be thrown	`dict_name.pop(<key>)`
update()	Updates the dictionary with another dictionary or a sequence of key-value pairs	`dict_name.update(<obj>)`
keys()	Returns a list of all keys in the dictionary	`dict_name.keys()`
values()	Returns a list of all values in the dictionary	`dict_name.values()`
items()	Returns a list of tuples with all (keys, values) pairs in the dictionary	`dict_name.items()`

For a complete list of dictionary methods see the Python official documentation here (https://docs.python.org/3/library/stdtypes.html#mapping-types-dict).

Control flow

We now turn our attention to control flow statements and constructs. As we will see, the sequence and mapping data types above will be tightly intertwined with loops and conditional statements.

Loops

There are cases in which our program consists of many repetitive tasks.

Let's say that we want to write a program that sequentially prints out the name of each person from a given list of names.

One alternative is the following:

In [15]:

```
# Define a list containg names
names = ["Luca", "Chiara", "Marco", "Gabriella", "Davide"]

# Sequentially print out the names from the list
print(names[0])
print(names[1])
print(names[2])
print(names[3])
print(names[4])
```

```
Luca
Chiara
Marco
Gabriella
Davide
```

As you can see in the program above, we are **repeating** the `print` command over and over. The only difference is that each time we are indexing the list to access the next name and print it to screen.

This is not a big deal with 5 names as above, but you can see how typing `print(names[index])` can be time consuming if we have 100 names or more in the list.

For loops

Luckily, Python is on our side. In programs similar to the one above and in many other situations, Python allows us to perform *repetitive tasks* by using a **for loop**.

> In programming, a **for loop** is a control flow statement that allows a portion of code to be executed iteratively.

In Python, the general syntax of a for loop is

```
for item in iterable:
    statement(s)
```

where

- `iterable` is any data type or expression that can return one `item` at a time
  - This is the sequence of elements we want to **iterate over**
  - List, tuples, and dictionaries are examples of iterables
- `item` is the object that is returned from the `iterable` at each iteration of the loop. This will be used in the `statement(s)` block.
  - This can be a single value, e.g. an element of a list, or multiple values, e.g. a tuple of values.
- `statement(s)` is the portion of code that is **repeated** at each iteration of the loop.
  - This can be a single statement in one line as well as a more complicated block of statements over multiple lines

Important: the portion of code that we want Python to evaluate iteratively within the loop **must be indented**, that is must be written with same number of leading whitespaces. The above definition reflects this.

Let's revisit our program that prints out names from a list of names. This can be easily done with a for loop as follows:

In [16]:

```python
# Print out the names from the list using a for loop
for name in names:
    print(name)
```

```
Luca
Chiara
Marco
Gabriella
Davide
```

Another alternative is to iteratively access each element in the list of names by indexing the list at each iteration of the loop.

In order to so we need to be able to create and iterate over a **sequence of integers** which we will use as indexes. This is done using the `range()` function.

In [17]:

```
# Iterate over a sequence of integers with range() and display the value at each iteration
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

As you can see, the for loop above exactly creates the integers we need to index all elements in the list iteratively, that is at each iteration of the loop.

To do that we simply need to index the list in the `statements` block of the for loop:

In [18]:

```
# Print out the names from the list by indexing the list directly
for i in range(5):
    current_name = names[i]
    print(current_name)
```

```
Luca
Chiara
Marco
Gabriella
Davide
```

While loops

Another way to iteratively execute a block of statements is by using a **while loop**.

> In programming, a **while loop** is a control flow statement that allows a portion of code to be executed iteratively **as long as a certain condition is met**.

In Python, the general syntax of a while loop is

```
while condition:
    statement(s)
```

where

- `condition` is **conditional expression** that is either `True` or `False` ,
  - `True` and `False` are known as Booleans, which is another data type in Python.
  - `condition` can be either a conditional expression that evaluates to a Boolean or a Boolean itself.
- `statement(s)` is a portion of code that is executed **only if** the `condition` is `True` .

In plain English, the while loop can be thought as the equivalent to "while condition is true, execute statements".

Important: variable initialisation

While loops always check whether the condition is met **before** entering the next iteration and executing the `statement(s)` block.

This applies to the very first iteration as well. Therefore, it is necessary to **initialise** any value contained in the conditional expression **before** the while loop.

While loops in action

Let's say that we want to write the following program:

> Given an initial number, keep doubling it until it reaches 100

In other words, **as long as** the number is below 100 the program should keep on doubling the number. Therefore, the `condition` is that the number should be **less than** 100 and the `statement` is the doubling of the number itself.

In [19]:

```python
# Initialise the number to a value and print it
number = 1
print(f"We started with {number}\n")

# Keep doubling the number until it reaches 100
while number < 100:
    # Double the number
    number *= 2          # This is equivalent to number = number * 2

    # Print to keep track of the update
    print(f"We are now at {number}")

# Print the final result
print(f"\nWe ended with {number}")
```

```
We started with 1

We are now at 2
We are now at 4
We are now at 8
We are now at 16
We are now at 32
We are now at 64
We are now at 128

We ended with 128
```

Conditional statements

Everyday we make decisions based on the outcome of particular event:

```
If it rains, I'll stay at home.

If I win this bet, I'll get £100; otherwise I'll lose £100.

If I win this bet, I'll go celebrate; otherwise I'll check my bank account
first before joining my friends at the pub.
```

In programming, the same concept applies. Often times we want to execute a series of instructions that **depend on** the result of previous instructions.

In Python, we can do that by using a combination of **conditional expressions** and `if..else` statements.

> A **conditional expression** is an expression that returns a Boolean value of either `True` or `False`
>
> An `if..else` statement is a **conditional statement** that executes a series of instructions depending on the value of a conditional expression.

Conditional expressions: True or False?

Conditional expressions allow us to verify whether a certain conjecture holds. When evaluated they return a Boolean value of `True` or `False` .

Comparison operators

If we want to compare two (or more) instances of the same data type we use **comparison operators**:

Operator	Meaning	Usage	Result
`==`	Equal	`x == y`	`True` if value of `x` is equal to value of `y` , otherwise `False`
`!=`	Not equal	`x != y`	`True` if value of `x` is not equal to value of `y` , otherwise `False`
`>`	Greater than	`x > y`	`True` if `x` is greater than `y` , otherwise `False`
`<`	Less than	`x < y`	`True` if `x` is less than `y` , otherwise `False`
`>=`	Greater than or equal to	`x >= y`	`True` if `x` is greater than or equal to `y` , otherwise `False`
`<=`	Less than or equal to	`x <= y`	`True` if `x` is less than or equal to `y` , otherwise `False`

Let's see a few examples:

In [20]:

```
4 == 5
```

Out[20]:

```
False
```

In [21]:

```
4 != 5
```

Out[21]:

```
True
```

In [22]:

```
4 < 5 < 7
```

Out[22]:

```
True
```

In [23]:

```
4 > 5 < 7
```

Out[23]:

```
False
```

The result of a conditional expression can be saved in a variable and used at a later stage. The data type of this variable will be a Boolean.

As the example below shows, we can compare strings as well:

In [24]:

```
# Check if capitalise and lowercase strings are the same
string_equality = "Coding" == "coding"

# Print result and data type
print(string_equality)
type(string_equality)
```

```
False
```

Out[24]:

```
bool
```

Boolean operators

Also known as logical operators, **Boolean operators** allow us to check multiple conditions at the same time.

Operator	Usage	Result
`and`	condition1 `and` condition2	`True` if both conditions are `True` , otherwise `False`
`or`	condition1 `or` condition2	`True` if at least one of the conditions is `True` , otherwise `False`
`not`	`not` condition	`True` if condition is `False` , and vice versa

In [25]:

```
2 < 3 and 4 < 5
```

Out[25]:

```
True
```

In [26]:

```
2 == 3 or 4 != 5
```

Out[26]:

```
True
```

In [27]:

```
not string_equality
```

Out[27]:

```
True
```

Membership operators

Finally, **membership operators** allows use to check whether a given element is present in a sequence of elements

In [28]:

```
"Luca" in names
```

Out[28]:

```
True
```

In [29]:

```
"econ" not in "economics"
```

Out[29]:

```
False
```

`if..else` statements

We are now ready to use conditional expressions to control which set of instructions gets executed. This is achieved using `if..else` statements.

In Python, the general syntax of an `if..else` statement is

```
if condition:
    <statements block 1>
else:
    <statements block 2>
```

where

- `<statements block 1>` is executed if `condition` is `True`
- `<statements block 2>` is executed if `condition` is `False`

The `else` block can be omitted, in which case nothing gets executed if `condition` is `False`. The program ignores the `if..else` altogether and continues to the next set of instructions.

Let's consider our real life example above. In plain English, we have

```
If it rains, I'll stay at home.
```

In Python, this is programmed as follows

In [30]:

```
# Set the weather
weather = "sunny"

# Choose an action depending on the weather
if weather == "rain":
    print("No thanks, I'll stay inside.")
else:
    print("Cool, let's go outside!")
```

```
Cool, let's go outside!
```

Or for instance

```
If I win this bet, I'll get £100; otherwise I'll lose £100.
```

becomes

```
In [31]:
```

```python
# Set bet result and income
won_bet = False
income  = 1000

# Update income depending on the result of the bet
if won_bet:
    income += 100
else:
    income -= 100

# Display final income
print(income)
```

```
900
```

Nested if statements: `if..elif..else`

Finally, consider our last example

```
If I win this bet, I'll go celebrate; otherwise I'll check my bank account
first before joining my friends at the pub.
```

Notice that there are **two** conditionals. The winning of the bet, and the amount of money in the bank account. These are considered sequentially.

First, I am checking whether I have won the bet

- If yes, I'll go celebrate straight away

Second, I am checking how much money I have in my account *in case I have lost the bet*

- If I have enough money, I'll go the pub anyway
- If I am broke, I'll stay at home

In Python, such cases are handled by **nested if statements**. The general syntax is

```
if condition1:
    <statements block 1>
elif condition2:
    <statements block 2>
else:
    <statements block 3>
```

where the `elif` block, short for "else if", allows us to check an additional condition and execute an additional set of instructions.

The two conditions, `condition1` and `condition2` , are considered sequentially:

- If `condition1` is `True` , then `<statements block 1>` is executed ***regardless*** of the value of `condition2` .
- If `condition1` is `False` , then the value taken by `condition2` determines whether `<statements block 2>` or `<statements block 3>` is executed.

In Python words, this is done as follows

In [32]:

```python
# Set bet result and bank account balance
won_bet = False
balance = 500

# Update bank account balance
balance += (100 if won_bet else -100)
print(f"As a result of the bet your balance is now £{balance}\n")

# Choose an action
if won_bet:
    print("I have won! Let's celebrate 🍻🎉 ")
elif balance > 300:
    print("I have lost but I have enough money. I'll have a pint anyway...")
else:
    print("I have lost and I am broke. Better stay at home.")
```

```
As a result of the bet your balance is now £400

I have lost but I have enough money. I'll have a pint anyway...
```